Specification and Reasoning about Shared Realizations: A Two-Tiered Approach

Yu-Shan Sun, Diego Zaccai, and Murali Sitaraman

Technical Report RSRG-14-02
School of Computing
100 McAdams Hall
Clemson University
Clemson, SC 29634-0974

July 2014

Specification and Reasoning about Shared Realizations: A Two-Tiered Approach

Yu-Shan Sun[1], Diego Zaccai[2] and Murali Sitaraman[1]

[1] School of Computing, Clemson University, Clemson, SC 29634, USA
{yushans, murali}@clemson.edu
[2] Department of Computer and Information Science, The Ohio State University, Columbus, OH, 43210, USA
zaccai.1@osu.edu

Abstract. The general idea of data abstraction is well understood and most formal specification methods support their development and use in some form or the other. However, object encapsulation through component development in modern programming languages remains a problem, because clients can violate the abstraction by accessing object internals through aliased object references. This paper presents a two-tiered approach. It discusses the specification and reasoning machinery necessary to prove that a realization with a shared data representation is correct with respect to an abstract specification, and shows how clients of such realizations can be verified using abstract interfaces alone. The paper illustrates the ideas with a detailed example in which a shared realization is employed to produce an (amortized cost) constant time implementation of an operation to copy a buffer.

Keywords: Formal specification, linked data structures, verification

1 Introduction

Reasoning about realizations in which a data representation is shared among objects is a challenging problem. The objective of this paper is to illustrate with a simple, yet motivating example the key ideas necessary for formal verification of shared realizations without requiring special logics or verification machinery. The paper additionally illustrates the use of a two-tiered approach with intermediate data abstractions to compartmentalize such verification. The approach helps hide a host of complexities in verifying such realizations in a lower tier.

While the ideas in this paper could be illustrated using any number of examples, we consider one that is common in software development: Use of shared data representation to improve performance. The specific problem we consider is efficient copying of a buffer. It is easy to create a normal mutable buffer (or queue) that has constant time operations such as `Enqueue` and `Dequeue` but requires linear time to copy. Alternatively, immutable buffers could be implemented to have constant-time copying through reference copying, but the performance of other operations would suffer due to the immutability of the underlying structures. In the solution presented in this paper all operations, including copying,

take amortized constant time. While this idea itself is not novel, the two-tiered approach for verification is.

The verification approach employs abstract interfaces to compartmentalize reasoning. Rather than implement the immutable queue structure directly, we have chosen to (i) represent and implement a queue with a pair of stacks and (ii) implement the stacks with a shared realization. This separation helps us illustrate how verification of an immutable queue implementation is vastly simplified; For part (i), the paper includes results from automated verification using the Ohio State RESOLVE compiler [1]. For part (ii) for the shared stack realization, we explain the specifications and assertions necessary for verification; while generation of verification conditions for this more complex implementation is automated, tool work for automated discharge of those conditions is not yet complete.

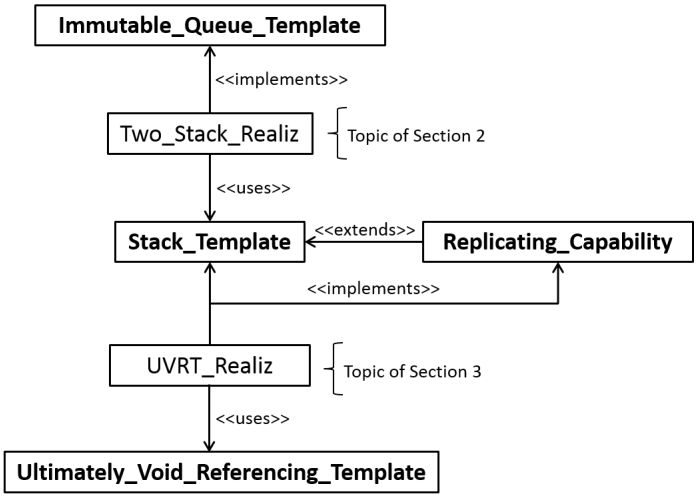

Fig. 1. An Illustrative Overview

Figure 1 illustrates the scope of the problem and the focus of the two tiers. (The text that is boldfaced indicates specification items, while the regular font indicate code realizations.) It is a UML diagram showing relationships among the various artifacts. Section 2 concerns the top tier. Starting from the top of the diagram, `Immutable_Queue_Template` is the specification of a queue concept that captures the spirit of an immutable queue. The figure also shows an implementation, named `Two_Stack_Realiz` for the queue concept. This implementation, named `Two_Stack_Realiz` is based on `Stack_Template`, and can be verified using only the specification of `Stack_Template` in a modular fashion automatically as outlined Section 2. The section also a contains a discussion of its constant-time performance behavior. Formal notations for writing performance

profiles (containing duration and memory usage estimates) that can serve as a basis for performance verification may be found elsewhere [2].

Section 3 concerns the lower tier. It discusses verification of the shared realization of `Stack_Template`, also in a modular fashion. This realization, named `UVRT_Realiz`, in Figure 1 uses `Ultimately_Void_Referencing_Template` (UVRT, for short) concept, a specialized version of the general pointering concept specified in [3]. The concept UVRT is constrained to allow only cycle-free pointer chains, the kind necessary for implementing data structures like stacks, queues, lists, and trees. The earlier paper also illustrates how describing pointer behavior through a formally-specified data abstraction concept makes it possible to use the same verification machinery for all realizations, whether they are built using typically built-in structures, such as pointers, or they are built using other data abstractions, such as stacks. Ultimately, the general pointering concept will be used to realize UVRT.

Section 4 of the paper contains related work and our conclusions are presented in Section 5.

2 Specification and an implementation of an Immutable Queue Concept

2.1 Immutable Queue specification

Figure 2 contains the specification of an immutable queue concept in RESOLVE notation [1]. An astute reader might notice that these queues are not strictly immutable: the value of one of the queues is "replaced" in every call to `Enqueue` or `Dequeue`. However, the design is similar in the spirit of a typical object-oriented immutable queue. Specifically, the operation to `Enqueue` an element into the queue results in two different queues, the original one and a new one that contains all of the elements from the original one plus the new element. Similarly, `Dequeue` does not alter the state of the queue from which the element is being removed, but instead preserves the original value of the queue and produces the element that is being removed and a new queue with the remaining elements in it. It is worth noting that the input queue on which the actions are being performed is restored, meaning that their values are unchanged; the output queue is specified to replace the second parameter queue that is passed by the caller.

```
Concept Immutable_Queue_Template (type Entry);
  Type Family Queue is modeled by Str(Entry)
    exemplar q;
    initialization ensures q = Empty_String;

  Operation Enqueue (restores q1: Queue, replaces q2: Queue,
                     clears x: Entry);
    ensures q2 = q1 o <#x>;
```

```
  Operation Dequeue (restores q1: Queue, replaces q2: Queue,
                     replaces x: Entry);
    requires q1 /= Empty_String;
    ensures q1 = <x> o q2;

  Operation IsEmpty (restores q: Queue): Boolean;
    ensures IsEmpty = (q = Empty_String);

end Immutable_Queue_Template;
```

Fig. 2. Specification of an Immutable Queue Contract

Some features of the language are worth mentioning here. First, the model types and values presented in specifications refer to a mathematical string of `Entry`, (mathematical type of the parameterized object). The `requires` clause always refers to input parameter values. In the `ensures` clause, the notation `#` in front of a parameter denotes the input value of the parameter. The `<_>` operator is a string constructor that creates a string containing only the element inside of it. The `o` operator denotes string concatenation.

2.2 Two-Stack realization of immutable queues

Our queue, similar to [4], is represented by two stacks: front and back. However, unlike [4], they are not immutable functional Lists, though to simulate immutability their values are not changed during the lifespan of a queue. The complete relationship between the abstract values of the stack and the abstract value of the queue they represent is provided by the correspondence (or abstraction function): `q.front o reverse(q.back)`. A sample two-stack representation of a queue and its corresponding conceptual value are shown in Figure 3.

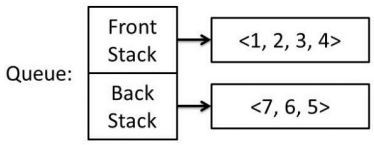

Fig. 3. A pictorial depiction of the abstract value of the front and back stacks in the representation of a `Queue` whose abstract value is `<1, 2, 3, 4, 5, 6, 7>`

The back stack contains the tail of the queue in reverse order (the last element enqueued will be in the top of the stack), while the front stack contains the front of the queue in order (the top of the stack should be the first element to be dequeued). Code for these procedures is given below. An abridged specification of `Stack_Template` on which this code is based is given following the implementation.

```
Procedure Enqueue (restores q1: Queue, replaces q2: Queue,
                   clears x: Entry);
  q2.front := Replica (q1.front);
  q2.back := Replica (q1.back);
  Push (q2.back, x);
end Enqueue;

Procedure Dequeue (restores q1: Queue, replaces q2: Queue,
                   replaces x: Entry);
  if not Is_Empty(q1.front) then
    q2.front := Replica (q1.front);
    q2.back := Replica (q1.back);
  else
    q1.front :=: q1.back;
    Reverse (q1.front);
    q2.front := Replica (q1.front);
    Clear (q2.back);
  end if;
  Pop (q2.front, x);
end Dequeue;
```

Fig. 4. Excerpts from a Two-Stack Implementation of the immutable queue

In a specification of stacks, `Stacks` are also conceptualized mathematically as strings of entries. Specifications of the operations mutating `Stack` operations `Push`, `Pop`, and `Replica` are straightforward and they are shown below.

```
Operation Push(updates S: Stack, clears E: Entry);
  ensures S = <#E> o #S;

Operation Pop(updates S: Stack, replaces E: Entry);
  requires S /= Empty_String;
  ensures #S = <E> o S;

Operation Replica(restores S: Stack): Stack;
  ensures Replica = S;
```

Fig. 5. Specification of Stack Operations

2.3 Automated Verification of Two-Stack Realization

In modular or specification-based verification, the implementation of a component is verified with respect to the contracts used in its representation, abstracting away all of their implementation details. The modularity of our proof system allow us to prove the correctness of our queue implementation using only the specifications used in its implementation, namely `Stack_Template` and `Replica`. The proof of the `Two_Stack` implementation in Figure 4 is completely independent of the shared implementation of `Stack_Template`. So the resulting verification conditions (VCs) (e.g., the one in Figure 6), involve only mathematical

string notations from stack and queue specifications and not pointer behavior. (Note: The λ symbol denotes `Empty_String`)

The verification of the implementation is done entirely automatically with the OSU tool-chain. The VC generator generated 12 verification conditions (VCs) for the code. Six of them were for `Dequeue`, 2 for `Enqueue`, 2 for the initialization, and 2 for `Is_Empty`. Most of the VCs fall into the "book keeping" category [5]. The OSU tool-chain allows the use of multiple verifiers and is also connected to Z3. However, due to the simplicity of the proofs, we decided to only use our in-house automatic verifier Split Decision [6]. All the VCs were verified.

$$
\begin{array}{ll}
1: & q1.front_0 \circ \text{reverse}(q1.back_0) \neq \lambda \\
2: \wedge & \langle x_{12} \rangle \circ q2.front_{12} = \text{reverse}(q1.back_0) \\
3: \wedge & |q1.front_0| \leq 0 \\
\hline
4: \Rightarrow & q1.front_0 \circ \text{reverse}(q1.back_0) = \\
& \langle x_{12} \rangle \circ q2.front_{12} \circ \text{reverse}(\lambda)
\end{array}
$$

Fig. 6. A verification condition for the correctness of `Enqueue`

Figure 6 shows perhaps the most interesting VC that arises in verifying the code in Figure 4 and is provided as an example of the simplicity of the proofs. From 3 we can deduce that `q1.front0` is empty. Given that, and the facts that the reverse of the empty string is itself the empty string, we can apply the result that the empty string is the identity for concatenation to simplify the goal in 4 to the given in 3.

2.4 Argument of constant-time performance behavior

As noted in [4], we claim that this queue has amortized constant-time performance for all of its procedures. The reasoning for this is as follows: elements can only be added to the queue by the `Enqueue` operation which makes calls to `Replica` and `Push`. It is not hard to see that `Push` could be implemented to work in constant time, and for now let us assume that so is `Replica`. (This is the topic of a later section.) The analysis of `Dequeue` has to be divided into two cases: If there is an element in the front queue then the performance argument is similar to that of `Enqueue`. However, if the front stack is empty, the act of reversing the stack is a linear time event. This is why the claim is for amortized constant time, notice that for an element to be dequeued it has to be moved from the back to the front stack. This will happen only once, thus the cost of reversing a stack with **n** items is distributed around **n** calls to `Dequeue`. There are ways to implement a constant time reversal for stacks, however those require the use of cycles in their representation's references and that would prevent us from using the cycle-free UVRT concept described in the next section, as well as hampering the mechanisms that allow us to claim constant time replica of a stack.

Using Copy on Write (COW) to maintain the illusion of having multiple copies of a mutable type, when in reality there is only one, is not a novel idea. For example, file systems have even implemented this to provide users with the impression that all users hold a unique copy of a mutable data-type even though the copies reside in the same place in the hard drive. Our technique for copying the stacks does not differ much from those with the exception that since the implementation in done at software's application level, the copying of objects can be finely tuned to match the needs of the data structure. Perhaps what is novel here is the use of COW systems to efficiently implement immutable types.

3 A Shared Realization of Stacks with UVRT

This section explains a shared realization of `Stacks` with suitable annotations. In showing the correctness of this realization, the following key points need to be made:

1. There are no cycles in the representation.
2. There are no memory leaks.
3. Updating a stack with a push or a pop does not affect the values of any other object.
4. Stack replica is done in (amortized) constant time.
5. Reference counting and Copy on Write provide a mechanism to satisfy 3-4 given 1-2.

In our design, the first two points come "for free" because they are encoded in the specification of the cycle-free pointering concept (UVRT) on which shared `Stack` implementation is based; this specification is the topic of Section 3.1. The third point is made in the discussion in Section 3.2. In that section, we also explain briefly (but not formally) how points 4 and 5 are achieved.

3.1 Ultimately Void Referencing Template

`Ultimately_Void_Referencing_Template` (UVRT) is a specialized version of a pointer concept that is especially suitable for implementing non-cyclic structures. The complete UVRT specification may be found in [7].

By creating an instance and using suitable operations, one can develop a singly linked list structure where the data in the nodes are of the actual type that is passed as the argument to the concept in instantiation. (The more general version of the concept includes the number of links per node as a parameter and is useful to implement trees.) In order to better explain UVRT specification, the concept has been broken down to smaller sections.

```
Concept Ultimately_Void_Referencing_Template(type Info);
  uses Function_Theory with Terminal_Range_Op_Ext;

  Defines Location: Set;
  Defines Void: Location;
```

```
Var Ref: Location -> Location;
Var Content: Location -> Info;
  constraints Terminal_Range(Location, {Ref}, Location) ⊆
    {Void} which_entails Ref(Void) = Void;
  initialization ensures Ref[Location] = {Void}
    and ...
```

Fig. 7. UVRT (Shared State)

For the formal function definitions and notations used in the specification, the concept in figure 7, makes use of `Function_Theory` and its extensions. In the figure, the `Location` set is an abstraction of the address space and its actual size is defined and constrained by an implementation on the underlying machine. `Void` is a special `Location`. A key idea in the concept is the use of two global state variables to capture shared state: `Ref`, a function that gives the "next" location for a given location and `Content`, a function that gives the information value referenced by a given location.

UVRT specification has been designed with the goal of enabling automated verification, though we have not achieved this goal as yet. Specifically, through carefully defined notations and theories, it avoids the use of quantifiers in assertions entirely.

Point 1: Absence of Cycles In figure 7, the key constraint is that for every location if we follow its next `Ref` chain will ultimately reference (or reach) the `Void` location. This constraint is the basis for the name for the concept and it is point 1 given at the beginning of Section 3. Since this constraint is already a given, when we implement Stack (or List or Tree) using UVRT, it becomes a freely established representation invariant that requires no further proof.

In order to express the constraint formally, we use a mathematical definition `Terminal_Range`. The general application of `Terminal_Range(U, {F, H}, G)`, where `U` is a set, `G` is a subset of `U` and `F` and `H` are functions, returns a set of elements that result from applying the functions `F` and `H` to the limit of each member of the set `G`. Figure 8 provides an illustration of this definition.

In the present constraint, there is only one function `Ref` that is applied to the set `Location` to determine the terminal range which is restricted to be just `Void`. The *which_entails* clause gives a lemma (that needs to be proved and) that becomes a useful lemma in the automated verification process.

In verifying shared `Stack` or other realizations that are based on UVRT, verification conditions involve `Terminal_Range`, and these will be discharged by an automated prover using pre-established theorems in `Terminal_Range_Op_Ext`.

When `UVRT` is instantiated, it ensures initially (only conceptually, of course) that all the locations reference `Void`. In this assertion, `Ref[Location]` denotes the set of range values that correspond to `Location`, a subset of `Ref`'s domain.

We have omitted constraint, initialization, and other assertions pertaining to the global state variable `Content`, because they are of less direct interest for this particular exposition.

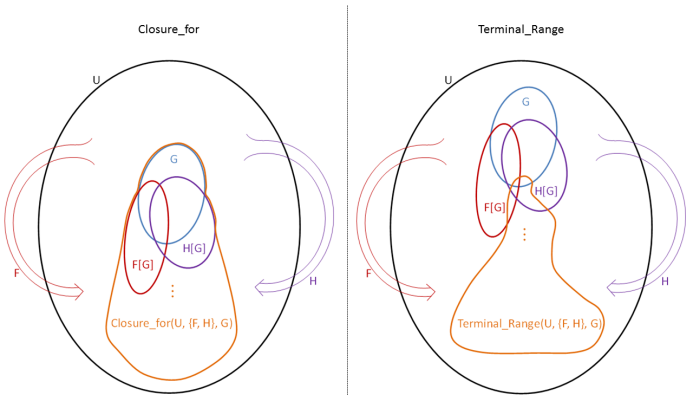

Fig. 8. Pictorial representation of `Closure_for` and `Terminal_Range`

Point 2: Absence of Memory Leaks In figure 9, UVRT defines a programming type `Pos` to represent a pointer, mathematically modeled as a `Location`. Initially, each position takes the value `Void`. (`exemplar` is just an example value of type `Pos`) A second key idea (concerning point 2) is "accessibility" and it is specified by a variable mathematical definition `Accessible_Loc`, whose value depends upon the global state variable `Ref`.

The formal definition of `Accessible_Loc` is based on a mathematical definition `Closure_for`, also defined and elaborated in `Terminal_Range_Op_Ext` theory. `Closure_for(U, {F, H}, G)` returns a set that results from applying the functions `F` and `H` repeated to the set of elements in `G`; `G` is a subset of `U`. Here, `Accessible_Loc` is the set of reachable locations produced by the `Closure_for` on all programming variables of a `Pos` type (i.e., all void-referencing pointer variables), unionized with `Void`.

In the definition of `Accessible_Loc` as well in the specification of operations, the following notations are used. They are a part of the specification language that allow us to make assertions about all objects or a specific object of a certain type or refer to the actual programming variable associated with a name.

- T.**Receptacles** denotes the set of all variables of type `T` that have been initialized, but not finalized
- **recp**.p is a specification language construct, it refers to the actual variable that will be associated with `p`
- **Val_in** (**recp**.p) denotes the mathematical value corresponding to the receptacle `p`

```
Type Family Pos is modeled by Location;
  exemplar p;
  initialization ensures p = Void;

  Def var Accessible_Loc: 𝒫(Location) = (
    {Void} ∪ Closure_for(Location, {Ref},
    Pos.Val_in[Pos.Receptacles]));

  finalization
    affects Ref, Content, Accessible_Loc;
    ensures ...
end;
```

Fig. 9. UVRT (Type Definition)

The finalization of a UVRT position variable (or pointer) – not shown for brevity – will have to deal with two scenarios. For all locations in `q` ∈ `Accessible_Loc` that are accessible from the set of all allocated locations minus `p` (the pointer that is being finalized or removed), then no changes are done to their references, i.e., `Ref(q) = Ref(#q)`. However, if some `q` is no longer accessible, because of the finalization of `p`, then `q` becomes available for allocation. In other words, every location is either available for allocation or is accessible, i.e., there are no memory leaks. In other words, the specification of UVRT demands that the underlying implementation of it do garbage collection. Using the new `Accessible_Loc`, the specification also states that that the `Content` prior to finalizing `p` is equal to the `Content` after finalization for accessible locations.

UVRT Operations The rest of the UVRT operations are discussed here briefly. `Give_New_Loc` allocates an unused location for a new `Pos`; it is the equivalent of memory allocation. `Redirect_Ref_at` makes referent point towards what `Ref(p)` points to. Operation `Follow_Ref` moves `p` to the reference pointed by `p`. Finalization for the original `p` will be in effect after this operation is called. `Swap_Content_of` swaps the information pointed at `p` with `I`. `Relocate_to` replace `p` with `New_L` and contents of old `p` is finalized; also unaccessible locations are specified to be free as in finalization. `Are_Colocated` checks if two `Pos` point to the same memory location. `Is_Almost_Inaccessible` checks if `p` can be accessible from other `Pos` other than `p`. `Is_Void` checks if a `Pos` is `Void`. `Set_to_Void` sets a `Pos` to `Void` and finalizes all resources. Figure 10 shows specifications for `Give_New_Loc` and `Redirect_Ref_at`, due to space constraints the rest of operations are ommited.

```
Operation Give_New_Loc (updates p: Pos);
  affects Accessible_Loc;
  requires p = Void;
  ensures p ∉ #Accessible_Loc;
```

```
Operation Redirect_Ref_at (preserves p: Pos,
                           updates referent: Pos);
  affects Ref;
  requires p ∉ Closure_for(Location,{Ref},{referent});
  ensures Ref = λq: Location.(
                  { #referent  if q = p
                  { #Ref(q)    otherwise
    ) and referent = #Ref(p);
```

Fig. 10. Give_New_Loc and Redirect_Ref_at operations for UVRT

3.2 A (simpler) shared realization of stacks without constant-time replication

An implementation of stacks with an amortized constant-time replica operation is sufficiently complex that a full exposition of that code is not possible within the constraints of this paper. So we first discuss a shared realization of a stack interface with only `Push`, `Pop`, and `Is_Empty` operations. This implementation shares the global variables in the instantiation of UVRT, and verification needs to ensure the frame property that the code modifies only the representation of its parameter `Stack` and nothing else; specifically, all other stacks must remain unchanged. An interesting aspect here is that the frame property verification is just a part of the process along with the specifications of stack operations as explained here.

In this implementation, the `Stack` is represented using a UVRT pointer position, which will require the creation of an instance of UVRT with the appropriate realization in the library. For each of the memory displacements, the actual space required is simply the amount of memory displacement required by creating a new `Position` as defined by `Ocpn_Disp_Incr` inside the implementation of UVRT. The *representation convention* states the set resulting from the `Closure_for` function with `S` intersected with the set resulting from the `Closure_for` function with all positions minus `S` (set difference is denoted by the symbol $\sim$) is simply just the set containing `Void`. This indicates that all locations are independent and are not shared. The *correspondence* (i.e., the abstraction function) takes a `Content` function as well as the `Ref` function and returns the sequence of `Entries`. In the correspondence, the $\prod$ symbol denotes iterated concatenation of a series of strings, whereas the notation $\mathtt{f}^n$ denotes application of function `f`, n times. For brevity, only the code for `Push` is shown in figure 11.

```
Realization Simple_UVRT_Realiz for Stack_Template;

  Facility UVR_Fac is
    Ultimately_Void_Referencing_Template(Entry)
    realized by Location_Referencing_Realiz;

  Type Stack = UVR_Fac.Pos;
    conventions ( Closure_for(Location, {Ref}, {S}) ∩
```

```
      Closure_for(Location, {Ref},
      Pos.Val_in[Pos.Receptacles ~ {recp.S}]) ⊆ {Void} );
    correspondence Conc.S =
                  ||Closure_for(Location, {Ref}, {S}|| - 1
                        ∏                    ⟨Content(Ref^(n-1)(S))⟩
                        n=1
  end;

  Procedure Push(updates S: Stack; clears E: Entry);
    affects Accessible_Loc, Content, Ref;
    ensures ...

    Var P : UVR_Fac.Pos;
    Give_New_Loc(P);
    Swap_Content_of(P, E);
    Redirect_Ref_at(P, S);
    Relocate_to(P, S);
  end Push;
          (* Code for other operations ommited *)
end Simple_UVRT_Realiz;
```

Fig. 11. An implementation of a Stack_Template using UVRT

Point 3: Ensuring Nothing Else Changes The operation `Push` affects the set of accessible locations as well as the content and references in the accessible locations. Since `Content`, `Ref`, and `Accessible_Loc` are shared variables and the RESOLVE language is based on clean semantics [8] (meaning only the explicit parameter objects are allowed to be modified), the *affects clause* raises a proof obligation that only the parameters are modified and nothing else. However, through the use of global state `Ref`, an implementation might change other `Content` of other stacks that are not parameters to `Push`.

This leads to a verification condition (an "internal" ensures clause) for `Push` to document how the internal shared variables are affected. It states that the accessible locations prior to calling `Push` are contained within the new set of accessible locations, the contents of all other accessible locations other than the actual variable associated with `S` remain the same and all the references of `q` have not changed if they were originally in the set produced by the `Closure_for` operation on all `Receptacles` of type `Pos` minus `S`. This frame property is established through a `which_entails` clause that follows the ensures clause and is shown in figure 12. (The ↾ restricts the domain of function `Val_in` to be the expression to the right of the symbol)

```
Stack.Val_in ↾ (Stack.Receptacles ~ {recp.S}) =
                  Stack.#Val_in ↾ (Stack.Receptacles ~ {recp.S})
```

Fig. 12. Frame property of `Push`

3.3 Outline of a shared realization with constant-time replication (Points 4 and 5)

The idea that a full deep copy of a structure with variable length is done in constant time should generate skepticism, and with good reason, since this is actually not possible. The trick for this is to hide from the clients the fact that we do not really make a copy when `Replica` is called, and then doing the actual copy only when the values of the items replicated are about to diverge. The implementation uses "Copy on Write" (COW) [6]. This implementation relies on reference counting as its main mechanism. Whenever a stack is copied with `Replica`, the reference count of that stack is increased. As an object comes out of scope, its reference count is decreased. The internal representation here takes the form (where `Record` is just a structure):

```
Type Data_w_Count = Record
  Data: Entry;
  Ref_Count: Integer
end;

Facility UVR_Fac is
  Ultimately_Void_Refng_Template(Data_w_Count)
  realized by Location_Referencing_Realiz;

Type Stack = UVR_Fac.Pos;
```

Fig. 13. The internal representation of a `Stack` using `Record` with reference counting

The constant-time `Replica` procedure is straightforward and it is as shown below, here the variable `Replica` represents the value the function's return value:

```
Operation Replica (restores S: Stack): Stack;
  ensures Replica = (S);
Procedure
  Var Temp: Data_w_Count;
  Swap_Content_of(S, Temp);
  Temp.Ref_Count := Temp.Ref_Count + 1;
  Swap_Content_of(S, Temp);
  Relocate_to(S, Replica);
end Replica;
```

Fig. 14. The body of `Replica` when using Copy on Write

An `Entry` is copied only when `Pop` is called on a stack that has previously been replicated. The representation conventions differ from the simpler one in that it doesn't restrict the intersection of the positions holding stack representations to be `Void`. The correspondence, however, is similar. Again, with respect to point #5, each procedure needs to ensure the frame property that when a `Stack` object is altered, none of the other stacks are modified. This is achieved by proving that whenever an object is modified, the number of references to its representation

is one. Whenever a procedure wants to modify an object whose reference count is larger than one, the component proceeds to do a "deep copy" of the object's representation before doing any modifications. It is important to note that this deep copy is not a real deep copy, since only the top level object is created, and the values inside of it are "copied" by calling replica. Because of this we say that this action just "pushes" the copy one level down into the representation.

As explained in [6], there are plenty of potential pitfalls with reference counting. Given RESOLVE's design, all of them can be divided into two categories: Unmanaged aliasing or Cycles in the references. The first of the problems is impossible in the RESOLVE language. The second one is not a possibility when using UVRT.

4 Related Work

The general difficulties and challenges in verifying shared realizations is the topic of [9]. This earlier research focuses on the principles and is a useful starting point. However, it does not address shared realizations based on pointer behavior, automation issues, or verification with layered constructs.

The closure results necessary for proofs in this work, such as reachability, are established independently. This factoring out of reusable mathematical development (independent of their application to the present verification problem) is a key reason for the simplicity of this treatment compared to, for example, [10]. Another key advantage of this method is that the logic used for the verification of the layered components and the pointer-based realizations is the same, there is no need for separate logics.

It would be interesting to study our approach to tree structures in relation to [11]. Our approach similarly involves establishing a mathematical theory of tree structures and using it to specify and reason about a Tree concept. However, the pointer-based implementation of the concept will be hidden (and verified once) using the UVRT templates described in this paper. Thus, such details will not routinely be raised in verification of client code.

The key principle of this research is to provide a way to reason about potential aliasing in the presence of references [12]. The verification system must provide a way to deal with aliasing across components when the programming languages allow references to be aliased as stated by Filipović, et al, Leavens, et al and O'Hearn, et al. [13][14][15]. There have been several different proposals to address this problem and Hatcliff, et al and Hogg, et al both presented summaries of these ideas in [16][17]. Separation logic and dynamic frames has emerged out of these ideas as two of the most promising in generating proofs involving references *within* a component [18][19]. In a more recent effort, region logic has been employed to translate from separation logic to dynamic frames, thus ensuring a single mechanism for reasoning about pointer behavior [20].

While we have focused on UVRT (a restricted version of pointers suitable for implementing a class of linked structures) in this paper, the two-tiered ver-

ification ideas can be generalized to general pointers [3], and various forms of memory management and garbage collection.

5 Conclusions

Verification of shared realizations based on pointer behavior remains a challenge. This paper has presented a two-tiered approach to concretize the ideas involved in such verification. In the process, we have presented a layered implementation of a persistent `Queue` (one that behaves like an immutable one despite changes to its underlying representation). We have shown that the layering allows us to write components that are relatively simple from a client perspective and amenable to modular verification, despite the rather complex nature of the entire composition. We have proved automatically the correctness of an `Immutable Queue` based on the contract of a `Stack`.

We have introduced a restricted form of references in the UVRT that can be built on top of a more general concept for pointers. The specification of UVRT is novel in many respects and it is designed to be automation friendly, using reusable mathematical developments, such as for closures to simplify verification. The restricted nature of the UVRT allows us to construct structures such as `Stacks` without a need to explicitly prove absence of cycles or memory leaks, avoiding the need for additional proofs. We have also introduced a list of properties we needed to prove to be able to create Stack representations that share parts of their representation with other stacks. Work is in progress to integrate the notations for verification of shared realizations into the compiler.

Acknowledgments. We thank members of the RESOLVE/Reusable Software Research Groups at Clemson and Ohio State for their inputs at various points in this research. Our special thanks are due to Joan Krone, Joe Hollingsworth, Bill Ogden, and Bruce Weide. This research is funded in part by U. S. National Science Foundation grants CCR-0113181, DMS-0701187, and CCF-1161916.

References

1. Sitaraman M., et al., "Building a Push-Button RESOLVE Verifier: Progress and Challenges", Formal Aspects of Computing 23 (3), Springer, 2011, 607-626.
2. Krone, J., Sitaraman, M., and Ogden, W.F., "Performance Analysis Based Upon Complete Profiles", Proceedings of SAVCBS 2006: FSE Workshop on Specification and Verification of Component-Based Systems, Portland, OR, Nov. 2006, 3-10.
3. Kulczycki, G., et al., "The Location Linking Concept: A Basis for Verification of Code Using Pointers". VSTTE'12 Proceedings of the 4th international conference on Verified Software: theories, tools, experiments, 2012, 34-49.
4. Burton, F. W. "An efficient functional implementation of FIFO queues". Information Processing Letters, 14(5):205206, July 1982.
5. Kirschenbaum, J., et al., "Verifying Component-Based Software: Deep Mathematics or Simple Bookkeeping?", Proceedings 11th International Conference on Software Reuse, Springer LNCS 5791, September 2009, 31-40.

6. Adcock, B. Working towards the verified software process, Ph.D. Thesis, The Ohio State University, 2010.
7. Sun, Y., Zaccai, D., Sitaraman, S.: Specification and Reasoning about Shared Realizations: An Illustrative Example. Technical Report RSRG-13-04, Clemson University (2013)
8. Kulczycki, G. "Direct Reasoning", Ph.D Dissertation, Clemson University, 2004.
9. Ernst, et al., "Modular Verification of Data Abstractions with Shared Realizations". IEEE Transactions on Software Engineering (TSE), Volume 20, Number 4, April 1994, 288-307
10. Lahiri, S., Qadeer, S. "Back to the future: revisiting precise program verification using smt solvers". In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 171182.
11. Wies, T., Muiz, M., Kuncak, V. "An Efficient Decision Procedure for Imperative Tree Data Structures". In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 476491. Springer, Heidelberg (2011)
12. Weide, B. W., and Heym, W.D. "Specification and verification with references". Proc. OOPSLA Workshop on Specification and Verification of Component-Based Systems, 2001.
13. Filipović, I., O'Hearn, P., Torp-Smith, N., Yang, H. "Blaming the client: on data refinement in the presence of pointers". Formal Aspects of Computing 22, 2010, 547-583.
14. Leavens, G. T., Leino, K. R. M., and Müller, P., "Specification and verification challenges for sequential object-oriented programs". Formal Aspects of Computing 19, 2007, 159-189.
15. O'Hearn, P., Reynolds, J., and Yang, H. "Local reasoning about programs that alter data structures". Proceedings 15th International Workshop on Computer Science Logic (CSL 2001), LNCS 2142, Springer, 2001, 1-19.
16. Hatcliff, J., et al., "Behavioral Interface Specification Languages". Dept. of EECS, University of Central Florida, CS-TR-09-01, March 2009.
17. Hogg, J., Lea, D., Willis, A., deChampeaux, D., and Holt, R. "The Geneva convention on the treatment of object aliasing". SIGPLAN OOPS Mess. 3, 1992, 11-16.
18. Gardner, P., and Zarfaty, U. "An introduction to context logic". Proc. Logic, Language, Information and Computation, LNCS 4576, Springer, 2007, 189-202.
19. Kassios, I. "Dynamic frames: support for framing, dependencies and sharing without restrictions". Proc. FM 2006: Formal Methods, LNCS 4085, Springer, 2006, 268-283.
20. Bao, Y., Leavens, G. T., and Ernst, G. "Translating Separation Logic into Dynamic Frames Using Fine-Grained Region Logic". Tecnical Report CS-TR-13-02a, Computer Science, University of Central Florida, Orlando, FL 32816, USA. 2014, 1-29.